06 - Divide and Conquer (part 1)

[KOMS119602] & [KOMS120403]

Design and Analysis of Algorithm (2021/2022)

Dewi Sintiari

Prodi S1 Ilmu Komputer
Universitas Pendidikan Ganesha

Week 14-18 March 2022
The principal of divide-and-conquer algorithm

Time complexity analysis of divide-and-conquer

Example of divide-and-conquer algorithms: MinMax problem

Divide-and-conquer based sorting
  - Merge Sort
  - Insertion Sort
  - Quick Sort
  - Selection Sort
Scheme of divide and conquer (DnC) algorithm
The principal of divide-and-conquer algorithm

**DIVIDE:** breaking down the problem into two or more sub-problems that have the same or similar type, until these become simple enough to be solved directly. Ideally, the size of the sub-problems are equal.

**CONQUER:** solving each of the sub-problems, directly (if the size is small) or recursively (if the size is still big).

**COMBINE:** combining the solutions to the sub-problems to produce a solution to the original problem.
The principal of divide-and-conquer algorithm

In the most typical case of divide-and-conquer, a problem’s instance of size \( n \) is divided into two instances of size \( n/2 \).

source: book of Anany Levitin
The principal of divide-and-conquer algorithm

source: https://cdn.kastatic.org/ka-perseus-images/db9d172fc33b90e905c1213b8cce660c228bb99c.png
Example of problems solvable by DnC algorithm

1. Merge sort
2. Quick sort
3. Closest pair problem
4. Convex hull problem (haven’t discussed yet)
5. Matrix multiplication
6. Strassen’s algorithm
7. Karatsuba algorithm for fast multiplication
8. Multiplication of two polynomials
Divide and conquer vs Brute force

Study case: sum of array of integers

Problem

*Given an array containing n integers* $a_0, a_1, \ldots, a_{n-1}$.

*Find* $a_0 + a_1 + \cdots + a_{n-1}$.

Brute-force approach? add the element sequentially (one-by-one)

Divide-and-conquer:

- If $n = 1$, then return $a_0$;
- If $n > 1$, then recursively do the following: divide into two sub-arrays, then compute the sum of each sub-array.

$$a_0 + a_1 + \cdots + a_{n-1} = (a_0 + \cdots + a_{\lfloor n/2 \rfloor-1}) + (a_{\lfloor n/2 \rfloor} + \cdots + a_{n-1})$$

Which technique is more efficient?
Divide and conquer vs Brute force

**Study case:** sum of array of integers

**Problem**

Given an array containing \( n \) integers \( a_0, a_1, \ldots, a_{n-1} \).

Find \( a_0 + a_1 + \cdots + a_{n-1} \).

**Brute-force approach?** add the element sequentially (one-by-one)

**Divide-and-conquer:**

- If \( n = 1 \), then return \( a_0 \);
- If \( n > 1 \), then recursively do the following: divide into two sub-arrays, then compute the sum of each sub-array.

\[
a_0 + a_1 + \cdots + a_{n-1} = (a_0 + \cdots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \cdots + a_{n-1})
\]

Which technique is more efficient?
The brute force technique is better in this case.
• DnC is probably the best-known general algorithm design technique.

• Not every divide-and-conquer algorithm is necessarily more efficient than (even) a brute-force solution.

• Often, the time spent on executing the DnC algorithm is significantly smaller than solving a problem by a different method.

• The DnC approach yields some of the most important and efficient algorithms in CS.
Algorithm 1 General scheme of divide-and-conquer

1: procedure DivideConquer(\(P\): problem, \(n\): integer)
2:     if \(n \leq n_0\) then ▶ P is small enough
3:         Solve \(P\)
4:     else
5:         Divide to \(r\) sub-problems \(P_1, \ldots, P_r\) of size \(n_1, \ldots, n_r\)
6:         for each \(P_1, \ldots, P_r\) do
7:             DivideConquer(\(P_i, n_i\))
8:         end for
9:         Combine the solutions of \(P_1, \ldots, P_r\) to solution of \(P\)
10:    end if
11: end procedure
DnC analysis of time complexity
Time complexity divide and conquer

\[ T(n) = \begin{cases} 
  g(n), & n \leq n_0 \\
  T(n_1) + T(n_2) + \cdots + T(n_r) + f(n), & n \geq n_0 
\end{cases} \]

- \( T(n) \): the time complexity of problem \( P \) (of size \( n \))
- \( g(n) \): time complexity for \texttt{SOLVE} if \( n \) is small (i.e. \( n \leq n_0 \))
- \( T(n_1) + T(n_2) + \cdots + T(n_r) \): time complexity to proceed each sub-problem
- \( f(n) \): time complexity to \texttt{DIVIDE} the problem and \texttt{COMBINE} the solution of each sub-problem
An ideal situation is when the Divide operation always produces two sub-problems of size half of the problem.

```plaintext
1: procedure DivideConquer(P: problem, n: integer)
2:   if n ≤ n₀ then ▷ P is small enough
3:     Solve P
4:   else
5:     Divide to 2 sub-problems P₁, P₂ of size n/2
6:     DivideConquer(P₁, n/2)
7:     DivideConquer(P₂, n/2)
8:     Combine the solutions of P₁, P₂ to solution of P
9:   end if
10: end procedure
```
If the instance is always divided into two sub-instances at each step, then:

\[
T(n) = \begin{cases} 
g(n), & n \leq n_0 \\
2T(n/2) + f(n), & n \geq n_0 \end{cases}
\]

More generally, if the instance is always divided into \( b \geq 1 \) instances of equal size, where \( a \geq 1 \) instances need to be solved, then the complexity is given by:

\[
T(n) = aT(n/b) + f(n)
\]

The order of growth of its solution \( T(n) \) depends on the values of the constants \( a \) and \( b \) and the order of growth of the function \( f(n) \).
MinMax Problem:
An example of DnC algorithm
MinMax problem (1)

Problem

*Given an array $A$ of $n$ integers. Find the min and max of the array simultaneously.*

Example:

| 4 | 10 | 21 | 11 | 23 | 3 | 42 | 34 | 1 |

$\text{min} = 1$

$\text{max} = 42$

Figure: An array of integers, and the min & max of the array
Algorithm 2 MinMax (brute-force)

1: procedure MinMax1(A[0..n − 1]: array, n: integer)
2:     min ← A[0]  
3:     max ← A[0]  
4:     for i ← 1 to n − 1 do
5:         if A[i] < min then
6:             min ← A[i]
7:         end if
8:         if A[i] > max then max ← A[i]
9:     end if
10:    end for
11: end procedure

▷ Assign the first element as the minimum
▷ Assign the first element as the maximum
The scheme of Minmax with divide-and-conquer

\[
\begin{array}{cccccccccccc}
4 & 10 & 21 & 11 & 23 & 3 & 42 & 34 & 1 \\
\end{array}
\]

**DIVIDE**

\[
\begin{array}{cccc}
4 & 10 & 21 & 11 \\
23 & 3 & 42 & 34 & 1 \\
\end{array}
\]

**CONQUER**: determine the min & max at each partition

- min = 4
- max = 21

- min = 1
- max = 42

**COMBINE**
Algorithm 3 MinMax (DnC)

1: procedure MinMax2(input: A, i, j, output: min, max)
2:     if i = j then min ← A[i]; max ← A[i]
3:     else
4:         if i = j − 1 then
6:             else min ← A[j]; max ← A[i]
7:         end if
8:     else
9:         k ← (i + j) div 2
10:        MinMax2(A, i, k, min1, max1)
11:        MinMax2(A, k + 1, j, min2, max2)
12:        if min1 < min2 then min ← min1
13:        else min ← min2
14:        end if
15:        if max1 < max2 then max ← max2
16:        else max ← max1
17:        end if
18:     end if
19: end if
20: end procedure
MinMax problem (5)

Example:

```
4 10 21 11 23 3 42 34

4 10 21 11
- min = 4
- max = 10

21 11
- min = 11
- max = 21

23 3 42 34
- min = 3
- max = 34

23 3
- min = 3
- max = 23

42 34
- min = 3
- max = 42
```

```
4 10 21 11 23 3 42 34

4 10
- min = 4
- max = 10

21 11
- min = 11
- max = 21

23 3 42 34
- min = 3
- max = 34

23 3
- min = 3
- max = 23

42 34
- min = 3
- max = 42
```
MinMax problem (6)

Example:

```
4 10 21 11 23 3 42 34 1
```

```
min = 4, max = 10
min = 11, max = 21
min = 3, max = 23
min = 42, max = 42
min = 1, max = 34
min = 1, max = 42
```
Compute the number of comparisons $T(n)$

\[
T(n) = \begin{cases} 
0 & \text{if } n = 1 \\
1 & \text{if } n = 2 \\
2 \cdot T(n/2) + 2 & \text{if } n > 2 
\end{cases}
\]

The explicit formula:

\[
T(n) = 2 \cdot T(n/2) + 2 \\
= 2 \cdot (2 \cdot T(n/4) + 2) + 2 = 4 \cdot T(n/4) + (4 + 2) \\
= 4 \cdot (2 \cdot T(n/8) + 2) + 4 + 2 = 8 \cdot T(n/8) + (8 + 4 + 2) \\
\vdots \\
= 2^{k-1} \cdot 1 + \sum_{i=1}^{k-1} 2^i \\
= 2^{k-1} + 2^k - 2 \\
= n/2 + n - 2 \\
= 3n/2 - 2 \in O(n)
\]
MinMax problem (8): Time complexity

- Brute force $\text{MINMAX1: } T(n) = 2n - 2$
- DnC $\text{MINMAX2: } T(n) = \frac{3n}{2} - 2$

$$\frac{3n}{2} - 2 < 2n - 2 \iff \text{ for } n \geq 2$$

The MinMax problem is more efficient to solve with DnC algorithm. But, asymptotically, both algorithms do not differ too much.
DnC-based sorting algorithms
DnC-based sorting (1)

Review

- **Sorting problem**: Given an ordorable array \( A[0..n−1] \) (of size \( n \)). The array \( A \) is sorted if the elements in \( A \) is ordered in an ascending or descending order.

- Recall that the brute-force-based sorting algorithms such as *selection sort*, *bubble sort*, and *insertion sort* have time complexity \( \mathcal{O}(n^2) \).

- Can we produce a sorting algorithm with a better time complexity using DnC approach?
Idea of DnC-based sorting procedures:

- If the array has size $n = 1$, then the array is sorted.
- If the array has size $n > 1$, then divide the array into two sub-arrays, then sort each sub-array.
- Merge the sorted sub-arrays into a sorted array. This is the result of the algorithm.
Divide and Conquer
Algorithm 4 DnC-based Sorting scheme

1: **procedure** `DnCSort(A[0..n-1]: array, n: integer)`
2: \hspace{1em} **if** `size(A) = 1` **then**
3: \hspace{2em} return `A`
4: \hspace{1em} **end if**
5: \hspace{1em} **Divide**(`A, A_1, A_2`) of size `n_1` and `n_2` resp. △ `n_2 = n - n_1`
6: \hspace{1em} `DnCSort(A_1, n_1)` △ `A_1 = A[0..n_1 - 1]`
7: \hspace{1em} `DnCSort(A_2, n_2)` △ `A_2 = A[n_1..n - 1]`
8: \hspace{1em} **Combine**(`A_1, A_2, A`)
9: **end procedure**

- **Divide** and **Combine** procedures depend on the problem.
Two approaches of DnC sorting algorithms

1. Easy split/hard join
   - The **Divide** step of the array is computationally easy
   - The **Combine** step is computationally hard
   - Examples: *Merge Sort, Insertion Sort*

2. Hard split/easy join
   - The **Divide** step of the array is computationally hard
   - The **Combine** step is computationally easy
   - Examples: *Quick Sort, Selection Sort*
DnC-based sorting (5)

Example

Given an array \( A = [4, 12, 3, 9, 1, 21, 5, 1] \)

1. **Easy split/hard join:** \( A \) is split based on the elements’ positions
   - *Divide:* \( A_1 = [4, 12, 3, 9] \) and \( A_2 = [1, 21, 5, 2] \)
   - *Sort:* \( A_1 = [3, 4, 9, 12] \) and \( A_2 = [1, 2, 5, 21] \)
   - *Combine:* \( A = [1, 2, 3, 4, 5, 9, 12, 21] \)

2. **Hard split/easy join:** \( A \) is split based on the elements values
   - *Divide:* \( A_1 = [4, 2, 3, 1] \) and \( A_2 = [9, 21, 5, 12] \)
   - *Sort:* \( A_1 = [1, 2, 3, 4] \) and \( A_2 = [5, 9, 12, 21] \)
   - *Combine:* \( A = [1, 2, 3, 4, 5, 9, 12, 21] \)
Merge Sort
Basic idea:

Merge Sort (1)

DIVIDE

A

4 10 21 11 23 3 42 34 1

CONQUER: sort each sub-array

A1

4 10 21 11

A2

23 3 42 34 1

4 10 11 21

1 3 23 34 42

COMBINE

A

1 3 4 10 11 21 23 34 42

Divide and Conquer
Merge Sort (2)

Algorithm:

Input: array $A$, integer $n$
Output: array $A$ sorted

1. If $n = 1$, then $A$ is sorted
2. If $n > 1$, then
   - Divide: split $A$ into two parts, each of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$
   - Conquer: recursively, implement MERGESORT in each sub-array
   - Merge: combine the sorted sub-arrays into the sorted array $A$
# Merge Sort (3)

## Algorithm 5 Merge Sort

1: **procedure** `MERGESORT`(`A`:ordable array, `i`, `j`: integer)  
   ▷ `i`: starting index, `j`: last index, initialization: `i = 0`, `j = n − 1` (i.e. the whole array `A`)
2:    **if** `i = j` **then**
3:       **return** `A[i]`
4:    **end if**
5:    `k ← (i + j) div 2`  
6:    `MERGESORT(A, i, k)`  
7:    `MERGESORT(A, k + 1, j)`  
8:    `MERGE(A, i, k, j)`  
9:    **end procedure**

![Diagram](A)
**Algorithm 6** “Merge” in **MERGESORT**

1: procedure **MERGE**(*A*, *i*, *k*, *j*) \(\triangleright \) *A*[\(i..k\)] and *A*[\(k+1..j\)] are sorted (ascending)

2: **output**: Array *A*[\(i..j\)] sorted (ascending)

3: **declaration**

4: \(B\) : temporary array to store the merged values

5: **end declaration**

6: \(p \gets i; \quad q \gets k + 1; \quad r \gets i\)

7: **while** \(p \leq k \text{ and } q \leq j\) **do** \(\triangleright \) while the left-array and the right-array are not finished

8: \[\text{if } A[p] \leq A[q] \text{ then}\]

9: \(B[r] \gets A[p]\) \(\triangleright \) *B* is a temporary array to store the merged array; assign *A*[\(p\)] (of left array) to *B*

10: \(p \gets p + 1\)

11: \[\text{else}\]

12: \(B[r] \gets A[q]\) \(\triangleright \) Assign *A*[\(q\)] (of right array) to *B*

13: \(q \gets q + 1\)

14: **end if**

15: \(r \gets r + 1\)

16: **end while** \(\triangleright \) At this point, \(p > k\) or \(q > j\)
1: \textbf{while }p \leq k \textbf{ do}  \quad \triangledown \textit{If the left-array is not finished, copy the rest of left-array A to B (if any)}
2: \quad B[r] \leftarrow A[p]
3: \quad p \leftarrow p + 1
4: \quad r \leftarrow r + 1
5: \textbf{end while}

6: \textbf{while }q \leq j \textbf{ do}  \quad \triangledown \textit{If the right-array is not finished, copy the rest of right-array A to B (if any)}
7: \quad B[r] \leftarrow A[q]
8: \quad q \leftarrow q + 1
9: \quad r \leftarrow r + 1
10: \textbf{end while}

11: \textbf{for }r \leftarrow i \textbf{ to } j \textbf{ do}  \quad \triangledown \textit{Assign back all elements of B to A}
12: \quad A[r] \leftarrow B[r]
13: \textbf{end for}
14: \textbf{return }A  \quad \triangledown \textit{A is in ascending order}
15: \textbf{end procedure}

\textbf{Remark. }the line numbering of the code is continued from the previous slide: 17, 18, 19, ...
Merge Sort (4): Procedure \textsc{Merge} example

\begin{figure}
\begin{center}
\begin{tikzpicture}
\node[rectangle, draw] (A) at (0,0) {4 10};
\node[rectangle, draw] (B) at (1,0) {21 11};
\node[rectangle, draw] (C) at (2,0) {4 10 11 21};
\node[rectangle, draw] (D) at (3,0) {3 23};
\node[rectangle, draw] (E) at (4,0) {3 23 42};
\node[rectangle, draw] (F) at (5,0) {1 3 23 34 42};
\node[rectangle, draw] (G) at (6,0) {1 3 4 10 11 21 23 34 42};
\node[rectangle, draw] (H) at (1,-1) {23 3};
\node[rectangle, draw] (I) at (2,-1) {42};
\node[rectangle, draw] (J) at (3,-1) {34 1};
\draw [->] (A) -- (B);
\draw [->] (B) -- (C);
\draw [->] (A) -- (H);
\draw [->] (B) -- (H);
\draw [->] (H) -- (I);
\draw [->] (I) -- (F);
\draw [->] (I) -- (J);
\end{tikzpicture}
\end{center}
\caption{Example of \textsc{Merge} procedure}
\end{figure}
Merge Sort (5): Procedure **MERGE_SORT** example

**Figure:** Example of **MERGE_SORT** procedure
The complexity of Merge Sort algorithm is measured from the number of comparisons of the elements in the array that is denoted by $T(n)$.

The number of comparisons is in $O(n)$, or $cn$ for some constant $c$.

(Here, we cannot compute exactly how many comparisons that we perform, because the Merge procedure involves many operations.)

So $T(n) = 2T(n/2) + cn$, for some constant $c$

Hence:

$$T(n) = \begin{cases} 0, & n = 1 \\ 2T(n/2) + cn, & n > 1 \end{cases}$$
The explicit function can be computed by iteratively substituting the function. For simplification, we compute the special case, when $n = 2^k$ for some integer $k$.

$$T(n) = 2T(n/2) + cn$$

$$= 2(2T(n/4) + cn) + 3cn$$

$$= 4(2T(n/8) + cn) + 3cn$$

$$\vdots$$

$$= 2^k T(n/2^k) + kcn$$

Since $n = 2^k$, then $k = \log_2 n$. This yields:

$$T(n) = n \cdot T(1) + cn \cdot \log_2 n = 0 + cn \cdot \log_2 n \in \mathcal{O}(n \log n)$$

This shows that Merge Sort has a better complexity ($\mathcal{O}(n \log n)$) than the brute-force-based sorting algorithms ($\mathcal{O}(n^2)$).
Recursive Insertion Sort

Special case of Merge Sort
This is an easy split/hard join-sorting.

We have seen an iterative version of Insertion Sort algorithm. We can also view it in a recursive way: it is a special case of Merge Sort.

The array is split into two sub-arrays, where the first sub-array only consists of one element, and the second sub-array consists of \( n - 1 \) elements.
Algorithm 7 Recursive Insertion Sort

1: procedure INSERTIONSORT($A$: ordorable array, $i, j$: integers)
2:  output: $A$ in ascending order
3:  if $i < j$ then
4:     $k ← i$
5:     INSERTIONSORT($A, i, k$)
6:     INSERTIONSORT($A, k + 1, j$)
7:     MERGE($A, i, k, j$)
8:  end if
9:  end procedure
**Remark.** Since the left sub-array is of size 1, then we may remove the \textsc{InsertionSort} procedure for the left sub-array.

\textbf{Algorithm 8} Insertion Sort

1: \textbf{procedure} \textsc{InsertionSort}(A: ordorable array, $i, j$: integers)
2: \hspace{1em} \textbf{output:} A in ascending order
3: \hspace{1em} \textbf{initialization:} $i \leftarrow 0$, $j \leftarrow n - 1$
4: \hspace{1em} \textbf{if} $i < j$ \textbf{then}
5: \hspace{2em} $k \leftarrow i$
6: \hspace{2em} \textsc{InsertionSort}(A, $k + 1, j$) \hspace{1em} ▷ \textit{A is split at position $i$ (initialize as $i = 0$}
7: \hspace{2em} \textsc{Merge}(A, $i, k, j$) \hspace{1em} ▷ \textit{sort the sub-array A[k + 1..j]}
8: \hspace{1em} \textbf{end if}
9: \textbf{end procedure}

\textbf{Remark.} The \textsc{Merge} procedure can be replaced with the 'Insertion method' used in the iterative version.
Example: Suppose that we want to sort the array
\[ A = [4, 10, 21, 11, 23, 3, 42, 34, 1]. \]

**Figure:** The 'Divide' and 'Conquer' steps
Figure: Applying the **MERGE** procedure
Insertion sort (6): Time complexity

The recursive formula for the TC:

\[
T(n) = \begin{cases} 
  a, & n = 1 \\
  T(n - 1) + cn, & n > 1 
\end{cases}
\]

The explicit formula is obtained by recursive substitution:

\[
T(n) = T(n - 1) + cn = (T(n - 2) + c(n - 1)) + cn = T(n - 2) + (cn + c(n - 1)) = T(n - 3) + (cn + c(n - 1) + c(n - 2)) \cdot \cdot \cdot \\
= cn + c(n - 1) + c(n - 2) + \cdots + 2c + a \\
= c \left( \frac{1}{2} \cdot (n - 1)(n + 2) \right) \\
= \frac{cn^2}{2} + \frac{cn}{2} + (a - c) \\
= \mathcal{O}(n^2) \quad \text{(same as in the iterative version)}
\]
Quick Sort

Click here
Recursive Selection Sort

Special case of Quick Sort
This is a hard split/easy join-sorting.

We have seen an iterative version of Selection Sort algorithm. We can also view it in a recursive way, as a special case of Quick Sort.

The array is split into two sub-arrays, where the first sub-array only consists of one element, and the second sub-array consists of $n - 1$ elements.

Remark. This method follows the Levitin’s version of **SelectionSort** (by looking for the min element). In the other version (if we look for the max element), the right sub-array has size one and the left sub-array has size $n - 1$. 
Remark. Since the left sub-array is of size 1, then we do not need to recursive call InsertionSort for the left sub-array.

Algorithm 9 Recursive Selection Sort

1: procedure SelectionSort(A: orderable array, \(i, j\): integers)
2: input: array \(A[i..j]\)
3: output: \(A[i..j]\) in ascending order
4: initialization: \(i \leftarrow 0, j \leftarrow n - 1\)
5: if \(i < j\) then
6: \(\triangleright \text{size}(A) > 1\)
7: Partition\((A, i, j)\) \(\triangleright\) Partition the array into sub-arrays of size 1 and \(n - 1\)
8: SelectionSort\((A, i + 1, j)\) \(\triangleright\) Sort only the right sub-array
9: end if
10: end procedure
Remark. Since the left sub-array is of size 1, then we do not need to recursive call \textsc{InsertionSort} for the left sub-array.

\textbf{Algorithm 10} Partition procedure

1: \textbf{procedure} \textsc{Partition}(A: ordorable array, $i, j$: integers) \hfill \triangleright

\textit{Partition} $A[i..j]$ by looking for the minimum element and assign it to $A[i]$

2: \hspace{1em} \text{idxMin} \leftarrow i

3: \hspace{1em} \textbf{for} $k \leftarrow i + 1$ \textbf{do to} $j$

4: \hspace{2em} \textbf{if} $A[k] < A[\text{idxMin}]$ \textbf{then}

5: \hspace{3em} \text{idxMin} \leftarrow k

6: \hspace{2em} \textbf{end if}

7: \hspace{1em} \textbf{end for}

8: \hspace{1em} \textsc{Swap}(A[i], A[\text{idxMin}]) \hfill \triangleright \text{Exchange } A[i] \text{ and } A[\text{idxMin}]

9: \textbf{end procedure}
Selection sort (4): Example

Suppose that we want to sort the array:
\[ A = [4, 10, 21, 11, 23, 3, 42, 34, 1] \]
Selection sort (4): Time complexity

The recursive formula for the TC:

\[
T(n) = \begin{cases} 
    a, & n = 1 \\ 
    T(n-1) + cn, & n > 1 
\end{cases}
\]

The explicit formula is obtained by substitution (as in Insertion Sort):

\[
T(n) = T(n-1) + cn \\
= (T(n-2) + c(n-1)) + cn = T(n-2) + (cn + c(n-1)) \\
= (T(n-3) + c(n-2)) + (cn + c(n-1)) = T(n-3) + \\
\quad (cn + c(n-1) + c(n-2)) \\
\vdots \\
= cn + c(n-1) + c(n-2) + \cdots + 2c + a \\
= c \left( \frac{1}{2} \cdot (n-1)(n+2) \right) \\
= \frac{cn^2}{2} + \frac{cn}{2} + (a-c) \\
= \mathcal{O}(n^2) \quad \text{(same as in the iterative version)}
\]
What can we conclude from the four sorting algorithms?

Splitting the array into two balanced arrays (of size \(n/2\) each) will result in the best algorithm performance (in the case of Merge Sort and Quick Sort, namely \(O(n \log n)\)).

While the unbalanced split (into 1 element and \(n - 1\) elements) results in poor algorithm performance (in the case of Insertion sort and Selection sort, namely \(O(n^2)\)).